UHD Worl
世界超高清

Audio Vivid 代码和工具应用指南

Guide to Audio Vivid Code and Tool Usage

UHD World Association
世界超高清视频产业联盟

UHD V
www.

前 言

本文件由 UWA 联盟组织制订，并负责解释。

本文件按照 GB/T 1.1—2020《标准化工作导则　第 1 部分：标准化文件的结构和起草规则》的规定起草。

本文件发布日期：xxxx 年 xx 月 xx 日。

本文件由世界超高清视频产业联盟提出并归口。

本文件主要起草单位：

世界超高清视频产业联盟、中央广播电视总台、腾讯科技有限公司、华为技术有限公司、字节跳动有限公司、马栏山音视频实验室、北京爱奇艺科技有限公司、湖南快乐阳光互动娱乐传媒有限公司、数字电视国家工程实验室（北京）、中移（杭州）信息技术有限公司、上海数字电视国家工程研究中心、荣耀终端股份有限公司、OPPO 广东移动通信有限公司

本文件主要起草人：

xxx、xxx

免责说明：

1，　本文件免费使用，仅供参考，不对使用本文件的产品负责。

2，　本文件的某些内容可能涉及专利，本文件的发布机构不承担识别这些专利的责任。

3，　本文件刷新后上传联盟官网，不另行通知。

目录

1. 范围

本文件规定了 Audio Vivid 音频编解码代码、FFMPEG 封装工具、ADM 转换工具、双耳渲染、扬声器渲染工具的应用方法。

本文件适用于 Audio Vivid 的内容制作、内容分发、内容回放领域。

2. 术语和定义

下列术语和定义适用于本文件。

1.位流 bitstream

用作数据编码表示的有一定次序的一组比特。

2.编码 coding

读入音频采样值并产生一个有效位流的过程。

3.编码器 coder

编码处理的实体。

4.编码位流 coded bitstream

音频信号的编码表示。

5.对象 object

被感知为一个整体的声音或由一个声源发出的独立于环境的声音。

6.解码 decoding

读入编码位流并输出音频采样值的过程。

7.解码器 decoder

解码处理的实体。

8.熵编码 entropy coding

信号数字表示中的一种变长无损编码，用以减少统计特性上的冗余。

9.声道 channel

声音在录制或播放时在不同空间位置采集或重放的相互独立的音频信号。

10.双声道立体声 stereo audio

一种音频格式，使用两个声道承载有一定相位关系或者幅度关系或者相位和幅度混合关系的音频信号，通常通过位于听音者前方的两个对称的扬声器重放，带给听音者更宽的声场感觉。

11.三维声 3D Audio

一种音频格式，多个声道承载构成完整音频内容的多路音频信号，通过环绕听音者的位于不同高度层的多个扬声器直接重放，或经过渲染或映射后重放，提供更高的声像空间解析度，并给听音者带来沉浸式的声场感觉。

12.元数据 metadata

描述音频数据的数据。

13.渲染 rendering

将给定的音频传输格式转换为适用于终端扬声器耳机配置的、可直接重放的音频格式的过程。

14.扬声器渲染 speaker rendering

将音频信号转换为特定配置的扬声器重放信号的过程。

15.双耳渲染 binaural rendering

将音频信号转换为双耳重放信号的过程。

3. Audio Vivid 端到端工具链和使用场景

3.1 工具链构成

Audio Vivid 标准端到端系统可以分为内容制作、内容分发、内容回放三个主要步骤。UWA Audio Vivid 标准技术体系提供了各个步骤需要使用到的代码和工具。主要涉及的工具以及对应的功能步骤如图 1 所示。

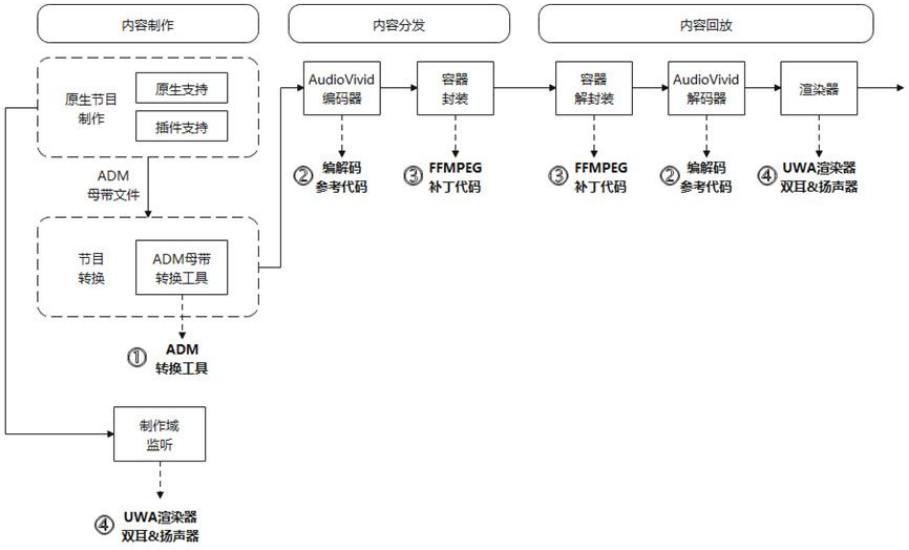

图 1 AudioVivid 标准端到端系统主要涉及工具及对应功能步骤

内容制作阶段可以分为原生节目制作和节目转换两种途径。原生节目制作指使用 DAW 和音频处理插件制作产出的原生 Audio Vivid 三维声节目。节目转换指从其他三维声格式，或从 ADM 母带文件转换得到

Audio Vivid 三维声节目。另外，节目制作阶段还需要制作域监听功能，以 DAW 为平台，在混音棚中进行节目渲染和监听。

内容分发阶段主要包括 Audio Vivid 编码和媒体容器封装两个主要步骤。Audio Vivid 编码器以音频文件和 Audio Vivid 元数据二进制文件为输入，输出 Audio Vivid ES 码流。媒体容器封装步骤以 Audio Vivid ES 码流为输入，根据所需媒体容器格式（如 MP4、TS、DASH 等）将 ES 码流进行封装。

内容回放阶段主要包括容器解封装、Audio Vivid 解码器、渲染器三个主要步骤。容器解封装是容器封装的逆过程，作用是从媒体容器格式中提取 Audio Vivid ES 码流和其他辅助信息。Audio Vivid 解码器以 ES 码流为输入，解码获得音频 PCM 数据和元数据数据结构。渲染器根据 PCM 数据和 Audio Vivid 元数据，将音频渲染到目标格式，如双耳或 5.1.4、7.1.4 等扬声器格式。

3.2 使用场景分析

Audio Vivid 标准和技术体系的使用者可以在上面描述的端到端系统中处于不同的位置，涉及其中部分或全部步骤。

本节根据 Audio Vivid 内容生产、分发、呈现生态系统中不同参与者的具体场景，给出各自涉及的模块和工具，如图 2 所示。

厂家类别	内容制作			内容分发		内容回放			工具编号
	制作工具	母带转换	监听	编码器	容器封装	容器解封装	解码器	渲染器	
节目制作	x		x						④
内容平台		x		x	x	x	x	x	①②③④
终端厂家						x	x	x	①②③
广电设备商			x	x	x				③②③

图 2 Audio Vivid 标准端到端系统涉及模块和工具

4. Audio Vivid 音频编解码标准参考代码

4.1 功能说明

Audio Vivid 编解码参考代码是 UWA Audio Vivid 音频编码标准的参考实现，算法描述请参考 UWA 标准《三维声技术规范第 1 部分：编码、分发与呈现》，标准编号 T/UWA 009.1-2023。

本参考代码除了 C 代码实现的 Audio Vivid 编解码算法外，同时包含针对 ARM 平台的解码器性能优化代码，可根据需要开启使用。

Audio Vivid 编解码参考代码下载链接：Audio Vivid 编解码参考代码

4.2 编译方式

Audio Vivid 参考代码提供两种编译方式，包括 Visual Studio 工程编译和 CMake 编译两种。以下分别给出使用方法。

1.Visual Studio工程

VS工程sln文件路径在AudioVivid_RM_1.1\AVS3_Codec\AVS3_Codec.sln，Visual Studio版本为VS2017。

VS工程Workspace中包含编码器和解码器两个工程，名称分别为avs3Encoder和avs3Decoder，可根据需要分别编译使用。

2.Cmake

CMakeLists 文件路径在 AudioVivid_RM_1.1\AVS3_Codec\CMakeLists.txt。

1) CmakeLists 内容和编译选项

CMakeLists 文件内容如下：

```
cmake_minimum_required(VERSION 3.5)
project(AVS3_baseline C)
set(CMAKE_C_STANDARD 11)
set(CMAKE_C_FLAGS "-Wall -O3 -ffp-contract=off")
include_directories(avs3Decoder/include)
include_directories(avs3Encoder/include)
include_directories(libavs3_common)
include_directories(libavs3_debug)
aux_source_directory(avs3Encoder/src encoder_src)
aux_source_directory(avs3Decoder/src decoder_src)
aux_source_directory(libavs3_common common)
add_executable(encoder ${encoder_src} ${common})
add_executable(decoder ${decoder_src} ${common})
```

add_library(av3adec SHARED ${decoder_src} ${common})

target_link_libraries(encoder m)

target_link_libraries(decoder m)

target_link_libraries(av3adec m)

注意：为保证 Audio Vivid 编码器和解码器中，基于 AI 模型的熵编码计算精度一致，避免出现解码信号杂音问题，需要在 CMakeLists 中打开如下编译选项（以上 CMakeLists 文件第 4 行）：

-ffp-contract=off

2) Cmake 编译方式

CMake 编译步骤如下：

Step 1：创建 build 目录：在 AVS3_Codec 目录下创建 build 目录。

Step 2：Cmake：进入 build 目录，执行命令 cmake ..。

Step 3：编译：执行命令 make -j。

3) 编译输出

本节所述 Cmake 文件编译后，可输出如下三个文件，分别是：

1、编码器可执行文件 encoder

2、解码器可执行文件 decoder

3、解码器算法库 libav3adec.so

3.ARM Neon优化选项

如前所述，Audio Vivid 编解码参考代码中包含了 ARM Neon 并行优化代码，其开启通过头文件中宏定义的开关控制。

ARM Neon 宏位置在 AVS3_Codec\libavs3_common 文件夹中 avs3_options.h 中，宏名称为 AVS_NEON_ON。

ARM Neon 优化宏默认关闭，如需在 ARM 平台编译使用，请开启此宏，并重新编译。

4.3 编解码工具使用说明

1.编码器命令行

1) 编译输出

Audio Vivid 编码器命令行形式如下：

avs3Encoder [options] [bitrate] [samplingRate] [inFileName] [outFileName]

命令行参数含义如下：

inFileName：输入文件名（*.wav）

outFileName：输出文件名（*.av3a）

bitrate：编码速率（bps）

samplingRate：输入信号采样率（kHz）。例如，对 44.1kHz 和 48kHz 采样率，该字段可分别配置为 44.1 和 48。

options 参数含义如下：

-nn_type: 神经网络模式配置，0 代表基本配置，1 代表低复杂度配置

-bitdepth：位深，16：16 比特的位深，24：24 比特的位深

-meta_file：(可选)元数据二进制文件，文件格式应符合标准文档中的元数据语法

-mono：单声道模式

-stereo：立体声模式

-mc channel_config：多声道模式，channel_config 为多声道扬声器配置，例如 MC_5_1_0 为 5.1 格式，MC_5_1_4 为 5.1.4 格式

-hoa order: HOA 模式，order 为 HOA 信号的阶数，1 为 FOA，2、3 分别对应 2 阶、3 阶 HOA

-mix soundBedType soundBed_channel_config soundBed_bitrate num_objs bitrate_per_obj: 混合信号模式。soundBedType 为声床类型，0 表示不包含声床（即纯对象信号），1 表示多声道声床+对象信号。soundBed_channel_config 为声床信号类型（支持立体声、5.1 等）。soundBed_bitrate 为声床信号编码速率。num_objs 为对象信号数量。bitrate_per_obj 为每个对象信号的编码速率。

2) 码率和扬声器配置说明

不同信号模式的可配置码率如表 1 所示：

表 1 Audio Vivid 可选码率表

信号模式	多声道模式（-mc）扬声器配置名称	混合模式（-mix）扬声器配置名称	码率（bps）
单声道/对象	不涉及	不涉及	32000, 44000, 56000, 64000, 72000,80000, 96000, 128000, 144000, 164000,192000
立体声	不涉及	STEREO	32000, 48000, 64000, 80000, 96000,128000, 144000, 192000, 256000,320000

信号模式	多声道模式（-mc） 扬声器配置名称	混合模式（-mix） 扬声器配置名称	码率（bps）
5.1	MC_5_1_0	MC_5_1_0	192000, 256000, 320000, 384000,448000, 512000, 640000, 720000,144000, 96000, 128000, 160000
7.1	MC_7_1_0	MC_7_1_0	192000, 480000, 256000, 384000,576000, 640000, 128000, 160000
5.1.2	MC_5_1_2	MC_5_1_2	152000, 320000, 480000, 576000
5.1.4	MC_5_1_4	MC_5_1_4	176000, 384000, 576000, 704000,256000, 448000
7.1.2	MC_7_1_2	MC_7_1_2	216000, 480000, 576000, 384000,768000
7.1.4	MC_7_1_4	MC_7_1_4	240000, 608000, 384000, 512000,832000
FOA	不涉及	不涉及	96000, 128000, 192000, 256000
HOA2	不涉及	不涉及	192000, 256000, 320000, 384000,480000, 512000, 640000
HOA3	不涉及	不涉及	256000, 320000, 384000, 512000,640000, 896000

说明：

1、码率表：各信号模式码率表和 UWA 编解码标准文档（T/UWA 009.1-2023）的对应关系如下：

单声道/对象码率表：表 A.10

立体声码率表：表 A.11

5.1 码率表：表 A.12

7.1 码率表：表 A.13

5.1.2 码率表：表 A.15

5.1.4 码率表：表 A.16

7.1.2 码率表：表 A.17

7.1.4 码率表：表 A.18

FOA 码率表：表 A.14

HOA2 码率表：表 A.19

HOA3 码率表：表 A.20

注意：混合模式 mix 情况下，每个对象的码率配置 bitrate_per_obj 应从表格 1 的单声道/对象行选择使用。

2、多声道模式：mc 模式可选信号格式包括 5.1、7.1、5.1.2、5.1.4、7.1.2、7.1.4，编码器命令行中 channel_config 字段应从表格 1 第 2 列中选择。

3、混合模式：mix 模式下，声床+对象配置时，声床可选的配置包括立体声、5.1、7.1、5.1.2、5.1.4、7.1.2、7.1.4，编码器命令行中 soundBed_channel_config 字段应从表格 1 第 3 列中选择。

3) 编码器命令行实例

各编码模式下，编码器命令行示例及说明如下：

示例 1：单声道编码

avs3Encoder -nn_type 1 -bitdepth 16 -mono 64000 48 test.wav test.av3a

-- 单声道编码，低复杂度配置，输入信号位深 16bit，采样率 48kHz，编码速率 64kbps，输入文件 test.wav，输出码流文件 test.av3a。

示例 2：立体声编码

avs3Encoder -nn_type 1 -bitdepth 16 -stereo 48000 48 test.wav test.av3a

-- 立体声编码，低复杂度配置，输入信号位深 16bit，采样率 48kHz，编码速率 48kbps，输入文件 test.wav，输出码流文件 test.av3a。

示例 3：多声道编码，5.1 格式

avs3Encoder -nn_type 1 -bitdepth 24 -mc MC_5_1_0 96000 48 test.wav test.av3a

-- 多声道编码（5.1 格式），低复杂度配置，输入信号位深 24bit，采样率 48kHz，编码速率 96kbps，输入文件 test.wav，输出码流文件 test.av3a。

示例 4：3 阶 HOA 编码

avs3Encoder -nn_type 1 -bitdepth 16 -hoa 3 256000 44.1 test.wav test.av3a

-- HOA 编码（3 阶），低复杂度配置，输入信号位深 16bit，采样率 44.1kHz，编码速率 256kbps，输入文件 test.wav，输出码流文件 test.av3a。

示例 5：混合模式编码，仅对象信号

avs3Encoder -nn_type 1 -bitdepth 16 -mix 0 4 44000 0 48 test.wav test.av3a

-- 混合模式编码（声床类型为 0，纯对象信号，对象数量 4），低复杂度配置，输入信号位深 16bit，采样率 48kHz，每个对象编码速率 44kbps，输入文件 test.wav，输出码流文件 test.av3a。

示例 6：混合模式编码，声床+对象信号

avs3Encoder -nn_type 1 -bitdepth 16 -mix 1 MC_5_1_0 192000 2 64000 0 48 test.wav test.av3a

-- 混合编码模式（声床类型为 1，声床类型 5.1，对象数量 2），低复杂度配置，输入信号位深 16bit，采样率 48kHz，5.1 声床编码速率 192kbps，每个对象编码速率 64kbps，输入文件 test.wav，输出码流文件 test.av3a。

示例 7：混合模式编码，声床+对象信号，包含元数据

avs3Encoder -nn_type 1 -bitdepth 16 -meta_file meta.bin -mix 1 MC_5_1_0 192000 2 64000 0 48 test.wav test.av3a

-- 混合编码模式（声床类型为 1，声床类型 5.1，对象数量 2），低复杂度配置，输入信号位深 16bit，元数据文件为 meta.bin，采样率 48kHz，5.1 声床编码速率 192kbps，每个对象编码速率 64kbps，输入文件 test.wav，输出码流文件 test.av3a。

4) 编码器运行依赖

Audio Vivid 编解码依赖标准规范的 AI 模型文件。

AI 模型文件为 AVS3_Codec\avs3Encoder 或 AVS3_Codec\avs3Decoder 文件夹下的 model.bin。两个文件夹下的模型文件完全相同。

编码器运行时，模型文件 model.bin 需要放置在编码器可执行文件的相同目录下。

2.解码器命令行

1) 解码器命令行参数说明

Audio Vivid 解码器命令行形式如下：

avs3Decoder [inFileName] [outFileName]

命令行参数说明：

inFileName：输入文件名（*.av3a）

outFileName：输出文件名（*.wav）

解码器命令行示例：

avs3Decoder test.av3a test_dec.wav

-- 对输入码流 test.av3a 进行解码，得到解码音频文件 test_dec.wav。

2) 解码器运行依赖

Audio Vivid 编解码依赖标准规范的 AI 模型文件。

AI 模型文件为 AVS3_Codec\avs3Encoder 或 AVS3_Codec\avs3Decoder 文件夹下的 model.bin。两个文件夹下的模型文件完全相同。

解码器运行时，模型文件 model.bin 需要放置在编码器可执行文件的相同目录下。

3) 解码器动态库使用方法

根据 5.2.2 节描述，Audio Vivid 参考代码使用 CmakeLists 形式编译时，可以同步获得解码器动态库。

若计划将解码器动态库链接到播放器进行使用，可以按如下方式处理。

解码器动态库依赖的头文件包括 avs3_stat_meta.h 和 avs3_dec_lib.h。其中，avs3_stat_meta.h 中定义了 Audio Vivid 标准的元数据结构，avs3_dec_lib.h 中给出了解码器动态库的接口函数定义和数据结构定义。

解码器动态库的调用方式可以参考标准参考代码中，解码器主调函数的实现方式，代码文件为 AVS3_Codec\avs3Decoder\src\decoder.c

若计划使用 FFMPEG 对接解码器，可参考 6.2.1 节中 FFMPEG 与解码器动态库的配合方式。

若需要使用静态库形式的解码器，可将 5.2.2 节中的 CMakeLists 文件稍作修改，将

```
add_library(av3adec SHARED ${decoder_src} ${common})
```

改为：

```
add_library(av3adec STATIC ${decoder_src} ${common})
```

4.4 解码器接口调用说明

参考代码中，解码器接口函数可以实现解码器实例初始化、码流头解析、码流帧解码、解码器关闭等功能。

1.解码器接口调用流程

调用解码器 API 实现解码过程的算法流程图如图 3 所示。

解码器接口函数声明在 libavs3_common/avs3_dec_lib.h 头文件中。

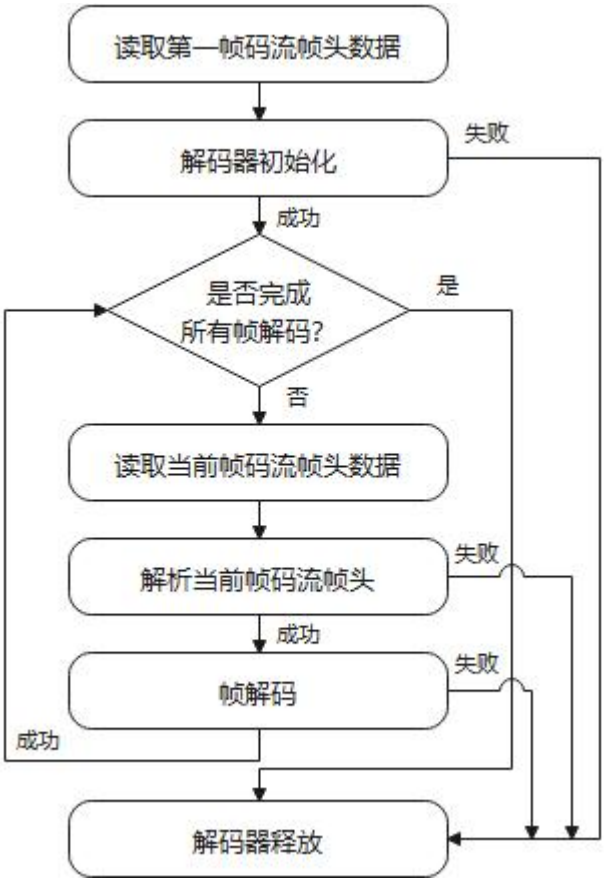

图 3 解码器 API 调用流程图

2.接口数据结构定义

1) 解码器句柄结构体

解码器句柄结构体定义如下：

```
struct Avs3DecoderLib {
    AVS3DecoderHandle hAvs3Dec;      // 解码器句柄
    uint16_t crcBs;                  // 码流中的 CRC 字段
    int16_t bytesPerFrame;           // 每帧码流长度(字节数目)
}
```

代码中结构体定义位置在 avs3Decoder/ avs3_dec_lib.c 中。

2) 解码器配置信息结构体

解码器配置信息结构体定义如下：

```
typedef struct Avs3DecoderLibConfigStruct {
    int32_t sampleRate;              // 采样率(Hz)
```

```
    int16_t numChsOutput;                    // 解码输出声道数量
}Avs3DecoderLibConfig
```

代码中结构体定义位置在 libavs3_common/avs3_dec_lib.h 中。

3) 元数据结构体

元数据结构体定义数量较多，具体请参考头文件 libavs3_common/ avs3_stat_meta.h。此处列出顶层元数据定义。

代码中的元数据定义与 T/UWA 009.1-2023 标准保持一致。

元数据顶层结构体定义如下：

```
typedef struct Avs3MetaDataStructure {
    int16_t hasStaticMeta;                          // 是否有静态元数据的标志
    int16_t hasDynamicMeta;                         // 是否有动态元数据的标志
    Avs3MetaDataStatic avs3MetaDataStatic;          // 静态元数据结构体
    Avs3MetaDataDynamic avs3MetaDataDynamic;  // 动态元数据结构体
}Avs3MetaData, *Avs3MetaDataHandle
```

静态元数据顶层结构体定义如下：

```
typedef struct Avs3MetaDataStaticStructure {
    int16_t hasVrExt;                               // 是否有 VR 扩展元数据的标志
    int16_t basicLevel;                             // 基础 ADM 元数据 Level
    Avs3BasicL1 avs3BasicL1;                        // 基础 ADM 元数据结构体
    int16_t vrExtLevel;                             // VR 扩展元数据 Level
    Avs3VrExtL1MetaData avs3VrExtL1MetaData;    // VR 扩展元数据结构体
}Avs3MetaDataStatic, *Avs3MetaDataStaticHandle
```

动态元数据顶层结构体如下：

```
typedef struct Avs3MetaDataDynamicStructure {
    int16_t dmLevel;                                // 动态元数据 Level
    int16_t numDmChans;                             // 对象信号声道数量
    int16_t muteFlag[32];                           // 对象信号静音标志
    int16_t transChRef[32];                         // 声道对应标志
    Avs3DmL1MetaData avs3DmL1MetaData[32];      // Level 1 动态元数据结构体
```

Avs3DmL2MetaData avs3DmL2MetaData[32]; // Level 2 动态元数据结构体

}Avs3MetaDataDynamic, *Avs3MetaDataDynamicHandle

3.接口数据结构定义

1) 解码器初始化接口

本接口用于创建解码器实例，接口定义如下：

int16_t Avs3DecoderLibCreate(

Avs3DecoderLibHandle * const hAvs3DecLib,

uint8_t *headerBs,

const char *modelPath)

接口功能为根据第一帧的码流帧头信息 headerBs、模型文件路径 modelPath 对解码器实例进行配置和初始化。

接口参数含义为：

a) Avs3DecoderLibHandle * const hAvs3DecLib：输出参数，解码器结构体指针。

b) uint8_t *headerBs：输入参数，第一帧的码流帧头信息，长度为 9 个字节。

c) const char *modelPath：输入参数，解码器依赖的模型文件路径。模型文件含义和路径见 1.3.2.2 节。

2) 解析帧头信息接口

本接口用于解析帧头信息，接口定义如下：

int16_t Avs3DecoderLibParseHeader(

Avs3DecoderLibHandle const hAvs3DecLib,

uint8_t *headerBs,

int16_t *rewind,

int16_t *bytesPerFrame)

接口功能为：对当前帧的帧头信息进行解析，获取解码器配置信息，以及当前帧码流的长度（字节数），以便读取比特流。

接口参数含义为：

a) Avs3DecoderLibHandle const hAvs3DecLib：输入参数，解码器结构体指针。

b) uint8_t *headerBs：输入参数，当前帧的码流帧头信息，长度为 9 个字节，实际有效长度为 7 个字节或 9 个字节（根据编码模式）。

c) int16_t *rewind：输出参数，表示码流需要回转的字节数，3DA 不同模式码流的帧头实际长度不同，变量表示在读取 9 个字节的帧头后，再进行回转的字节数。

d) int16_t *bytesPerFrame：输出参数，表示当前帧码流长度的字节数，用于解码接口读取码流数据。

3) 解码处理接口

本接口用于解码一帧音频码流，接口定义如下：

```
int16_t Avs3DecoderLibProcess(
    Avs3DecoderLibHandle const hAvs3DecLib,
    uint8_t *payload,
    int16_t *data,
    Avs3MetaDataHandle avs3MetaData)
```

接口功能为：以一帧码流数据为输入，获得解码后的音频数据（PCM，包括声床和对象）和元数据数据结构。

接口参数含义为：

a) Avs3DecoderLibHandle const hAvs3DecLib：输入参数，解码器结构体指针。

b) uint8_t *payload：输入参数，码流载荷指针，其中包括从码流中读取得到的一帧码流数据。

c) int16_t *data：输出参数，解码音频数据指针，包含交织形式存储的解码音频数据。

d) Avs3MetaDataHandle avs3MetaData：输出参数，解码 Audio Vivid 元数据结构指针。

4) 解码器关闭接口

本接口用于关闭并释放解码器实例，接口定义如下：

```
int16_t Avs3DecoderLibClose(
    Avs3DecoderLibHandle * const hAvs3DecLib)
```

接口功能为：关闭解码器实例，释放为解码器实例分配的内存。

接口参数含义为：

Avs3DecoderLibHandle * const hAvs3DecLib：输入/输出参数，解码器结构体指针。

5) 解码器配置获取接口

本接口用于获取解码器配置信息，接口定义如下：

```
int16_t Avs3DecoderLibGetConfig(
    Avs3DecoderLibHandle const hAvs3DecLib,
```

Avs3DecoderLibConfig *hAvs3DecLibConfig)

接口功能为：从解码器实例中获取解码器配置信息，包括采样率、解码输出声道数（解码器配置参数可根据需要扩充）。

接口参数含义为：

Avs3DecoderLibHandle const hAvs3DecLib：输入参数，解码器结构体指针。

Avs3DecoderLibConfig *hAvs3DecLibConfig：输出参数，解码器配置结构体指针，调用接口后解码器配置信息被赋值到数据结构中。

6) 辅助接口

辅助接口主要包括 wav 文件开启、写入数据和更新文件头信息等功能。

三个辅助接口定义如下：

Wav 文件打开接口：

```
FILE *Avs3DecoderLibOpenWavFile(
    Avs3DecoderLibHandle const hAvs3DecLib,
    const char* fileName)
```

写入数据接口：

```
void Avs3DecoderLibWriteWavData(
    Avs3DecoderLibHandle const hAvs3DecLib,
    const int16_t* data,
    FILE* fOutput)
```

更新 Wav 文件头信息接口：

```
void Avs3DecoderLibUpdateWavHeader(
    FILE* fOutput)
```

4.5 FAQ

Q1：编码器/解码器执行时，报 Can not open model file.错误

A：模型文件 model.bin 未放置在编码器/解码器同级目录下，或模型文件文件名不是 model.bin。

Q2：解码器输出的元数据数据结构如何获得，元数据结构在哪里定义？

A：解码器调用（参照 decoder.c）中，可通过 Avs3DecoderLibProcess()接口获得解码后的元数据，变量名为 avs3Metadata。元数据数据结构定义在头文件 avs3_stat_meta.h 中。

5. FFMPEG 封装工具

5.1 功能说明

FFMPEG 封装工具的作用是将 Audio Vivid 编码器输出的 ES 码流封装到指定的媒体容器格式中，例如 MP4、MOV、TS、DASH 等，以及反过程，即从媒体容器中解封装获得 ES 码流。同时，FFMPEG 补丁代码提供了与第 4 章解码器对接的能力，便于端侧播放器部署使用。

FFMPEG 封装工具的实现符合 UWA 联盟标准《三维声技术规范 第 2-2 部分 应用指南 媒体格式》，标准编号 T/UWA 009.2-2-2025。

FFMPEG 封装工具对应的代码以 FFMPEG 开源代码的补丁形式提供，基础 FFMPEG 版本为 n4.4.2。基础版本可从 FFMPEG 官方网站或 GitHub 下载获取。如需要基于其他版本的 FFMPEG 进行开发，则需要考虑版本间的差异问题，做必要的适配修改。

FFMPEG 封装工具分为两个版本，版本 1 仅包含媒体容器的封装和解封装功能，版本 2 在封装和解封装基础上，增加了对接 UWA 参考代码解码器接口的功能。

以下分别介绍两个版本的使用方法。

5.2 仅封装功能版本

此版本仅包含主流媒体容器的封装和解封装功能。

FFMPEG 封装工具仅封装功能版本下载链接：AUDIO Vivid 封装工具参考代码(基于 ffmpeg)

1.编译方式

1) 补丁构成和应用方式

补丁代码中包含在 FFMPEG n4.4.2 版本上，支持 AudioVivid 格式封装需要修改的所有代码和配置文件。

在基础版本 FFMPEG 4.4.2 版本基础上，将代码补丁中包含的代码文件覆盖到目录中即可。

补丁中包含的代码文件包括：

libavcodec 文件夹：av3a.c, av3a.h, av3a_parser.c, codec_dest.c, codec_id.h, Makefile, parsers.c, utils.c。

libavformat 文件夹：allformats.c, av3adec.c, isom_tags.c, Makefile, mov.c, movenc.c, mpegts.c, mpegts.h, mpegtsenc.c, rawenc.c。

2) 编译步骤说明

FFMPEG 工具编译需要 Linux 环境，以下给出编译步骤。

若计划使用单独的 FFMPEG 工具（即不将 FFMPEG 中的动态库单独编译），可按以下步骤编译。

Step 1：按 5.2.1 节中“补丁构成和应用方式”的说明，将补丁中的代码文件覆盖到 ffmpeg 目录下。

Step 2：（可选）如需支持汇编编译，可以安装 yasm 库，命令如下：

sudo apt-get install yasm

Step 3：修改编译工具和脚本的系统权限

chmod 777 configure ffbuild/*

Step 4：配置 ffmpeg

执行./configure

Step 5：编译

执行 make -j

若计划以 FFMPEG+动态库形式使用，即将 FFMPEG 中的 libavcodec 等单独编译动态库，则可以将上述步骤中的第 4 步替换为：

Step 4：配置 ffmpeg

执行./configure --enable-shared --disable-static

即使能动态库方式，禁用静态编译。

此时，ffmpeg 工具使用依赖编译过程中产生的如下动态库：

libavcodec.so.58

libavdevice.so.58

libavfilter.so.7

libavformat.so.58

libavutil.so.56

libswresample.so.3

libswscale.so.5

使用 ffmpeg 工具时，需要将上述 so 文件放置在 ffmpeg 工具同级目录下，并将当前目录增加到 LD_LIBRARY_PATH 环境变量中，命令如下：

export LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH

2.FFMPEG 工具命令行

当前版本 FFMPEG 工具主要支持 MPEG2 TS 和 MP4 类型的媒体容器格式。

将 Audio Vivid ES 码流封装到 TS 格式的命令行示例如下：

```
ffmpeg -i input.av3a -c copy output.ts
```

从 TS 格式中解封装获得 Audio Vivid ES 码流的命令行示例如下：

```
ffmpeg -i input.ts -c copy output.av3a
```

其中，input.av3a 和 output.av3a 分别是封装前和解封装后的 Audio Vivid ES 码流文件，output.ts 和 input.ts 为 TS 格式封装的媒体文件。

将 Audio Vivid ES 码流封装到 MP4 格式的命令行示例如下：

```
ffmpeg -i input.av3a -c copy output.mp4
```

从 MP4 格式中解封装获得 Audio Vivid ES 码流的命令行示例如下：

```
ffmpeg -i input.mp4 -c copy output.av3a
```

其中，input.av3a 和 output.av3a 分别是封装前和解封装后的 Audio Vivid ES 码流文件，output.mp4 和 input.mp4 为 MP4 格式封装的媒体文件。

5.3 封装+解码接口版本

此版本包含主流媒体容器的封装和解封装功能，以及与 UWA 参考代码解码器接口对接的功能，可一次性完成解封装和解码处理，获得解码后的音频文件。

FFMPEG 封装工具封装+解码接口版本下载链接：Audio Vivid 封装及解码工具（基于 ffmpeg）的参考代码

1.编译方式

1) 补丁构成和应用方式

补丁代码中包含在 FFMPEG n4.4.2 版本上，支持 AudioVivid 格式封装和解码器接口功能所需要修改的所有代码和配置文件。

在基础版本 FFMPEG 4.4.2 版本基础上，将代码补丁中包含的代码文件覆盖到目录中即可。

补丁中包含的代码文件包括：

FFMPEG 主目录下：configure

libavcodec 目录下：allcodecs.c, av3a.c, av3a.h, av3a_parser.c, avs3_dec_lib.h, avs3_stat_meta.h, codec_desc.c, codec_id.h, libav3adec.c, Makefile, parsers.c, utils.c

libavformat 目录下：allformats.c, av3adec.c, isom_tags.c, Makefile, mov.c, movenc.c, mpegts.c, mpegts.h, mpegtsenc.c, rawenc.c

2) 编译步骤说明

FFMPEG 工具编译需要 Linux 环境，以下给出编译步骤。

若计划使用单独的 FFMPEG 工具（即不将 FFMPEG 中的动态库单独编译），可按以下步骤编译。

Step 1：按 5.3.1 节中“补丁构成和应用方式”说明，将补丁中的代码文件覆盖到 ffmpeg 目录下。

Step 2：在 FFMPEG 目录下创建目录./bin/，将 4.2.2 节中“编译输出”编译获得的解码器算法库文件复制到此文件夹内。

解码器算法库可以是动态库或静态库，文件名形式为 libav3adec.*。Linux 平台下后缀名为 so 或 a。

注意：若解码器算法库为动态库形式，则需要将其路径./bin/添加到环境变量 LD_LIBRARY_PATH 中。

Step 3：在 FFMPEG 目录下创建目录./model/，将 Audio Vivid 标准中用到的模型文件 model.bin 复制到此文件夹内。

model.bin 文件位置可参考 4.3.1 节中“编码器运行依赖”或 4.3.2 节中“解码器运行依赖”所述。

Step 4：（可选）如需支持汇编编译，可以安装 yasm 库，命令如下：

```
sudo apt-get install yasm
```

Step 5：修改编译工具和脚本的系统权限

```
chmod 777 configure ffbuild/*
```

Step 6：配置 ffmpeg

执行./configure --enable-gpl --enable-libav3adec

--enable-libav3adec 选项表示开启 AudioVivid 解码器功能。

Step 7：编译

执行 make -j

若计划以 FFMPEG+动态库形式使用，即将 FFMPEG 中的 libavcodec 等单独编译动态库，则可以将上述步骤中的第 6 步替换为：

Step 6：配置 ffmpeg

执行./configure --enable-gpl --enable-libav3adec --enable-shared

即使能动态库方式。

此时，ffmpeg 工具使用依赖编译过程中产生的如下动态库：

libavcodec.so.58

libavdevice.so.58

libavfilter.so.7

libavformat.so.58

libavutil.so.56

libpostproc.so.55

libswresample.so.3

libswscale.so.5

使用 ffmpeg 工具时，需要将上述 so 文件放置在 ffmpeg 工具同级目录下，并将当前目录增加到 LD_LIBRARY_PATH 环境变量中，命令如下：

export LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH

2.FFMPEG 工具命令行

当前版本 FFMPEG 工具主要支持 MPEG2 TS 和 MP4 类型的媒体容器格式。

1) 封装功能命令行

封装和解封装功能的命令行形式与 5.2.2 节“FFMPEG 工具命令行”所述基本相同，此处仅以 MP4 格式为例说明差异部分。

将 Audio Vivid ES 码流封装到 MP4 格式的命令行示例如下：

ffmpeg -av3a_model model_path -i input.av3a -c copy output.mp4

从 MP4 格式中解封装获得 Audio Vivid ES 码流的命令行示例如下：

ffmpeg -av3a_model model_path -i input.mp4 -c copy output.av3a

其中，input.av3a 和 output.av3a 分别是封装前和解封装后的 Audio Vivid ES 码流文件，output.mp4 和 input.mp4 为 MP4 格式封装的媒体文件。

与 5.2.2 节“FFMPEG 工具命令行”所述命令行的差异：增加-av3a_model model_path 选项，其中 model_path 为模型文件路径。

2) 解码功能命令行

解码功能的作用是，以 MP4、TS 等格式的媒体容器文件为输入，完成解封装和 Audio Vivid 解码功能，获得解码后的 WAV 文件。

以 MP4 格式为例，给出解码功能命令行示例，如下：

ffmpeg -av3a_model model_path -i input.mp4 output.wav

其中，input.mp4 为 MP4 格式封装的媒体文件，output.wav 为 Audio Vivid 解码后的音频文件。

6. ADM 转换工具

6.1 功能说明

ADM 转换工具的功能是将符合 ITU-R BS.2076 标准的 ADM 音频母带文件，转换为可用于 Audio Vivid 编码使用的音频 wav 文件和元数据二进制文件。

配合 Audio Vivid 编码器可执行文件，可一次性完成 ADM 母带转换和 Audio Vivid 编码功能，获得 av3a 格式的码流文件。

当前 ADM 转换工具为二进制可执行文件形式，源代码暂未开源。

ADM 转换工具 exe 文件获取方式：发送邮件至 audio_vivid_support@theuwa.com，邮件主题：ADM 转换工具申请 + UWA 联盟会员名称 （请替换 UWA 联盟会员名称 为贵会员单位名称），收到此邮件后，联盟技术支持团队会及时回复并发布最新 ADM 转换工具。

6.2 编译方式

ADM 转换工具提供 Windows 和 Linux 两个版本，其编译工具链情况如下：

Windows 版本：CLion，MinGW，GCC 13.1.0，CMake 3.26.4

Linux 版本：Ubuntu 18.04，GCC 7.5.0，CMake 3.29.3

若工具使用过程中出现依赖问题，请检查上述编译工具链情况。

6.3 工具使用说明

ADM 转换工具命令行示例如下：

```
bw64_to_av3a  input  wavfile  metadata  [options]
```

必选参数含义如下：

input：输入 ADM 母带文件，需要是符合 BW64 ADM wav 规范的文件格式。

wavfile：输出音频文件，通用 wav 格式，满足 Audio Vivid 编码器声道数要求（即总声道数不超过 16）。

metadata：输出元数据二进制文件，符合 Audio Vivid 编码器格式要求。

可选参数含义如下：

-t xxx.txt，--txt xxx.txt：指定输出元数据可视化文件路径，文件名 xxx.txt，txt 格式，按 Audio Vivid 元数据数据结构展开打印，可用于定位元数据转换问题。

-m x，--max_channel x：用于指定输出 wav 文件的最大声道数为 x。其最大值不超过 16，最小值应等于声床声道数。即若声床为 7.1.2，则此参数最小值为 10。

-l xx，--target_loudness xx：用于指定输出音频节目的响度水平为 xx，响度单位为 LUFS/LKFS。

-a xxx.av3a，--av3a_out xxx.av3a：用于指定编码器输出码流文件，文件名为 xxx.av3a。配置此参数时将调用编码器完成 Audio Vivid 编码处理。

-e xxx，--encoder_path xxx：用于指定编码器可执行文件路径，可执行文件路径为 xxx。默认编码器可执行文件名为 encoder，与转换工具同路径。若编码器可执行文件路径为非默认，需要配置此参数。注意，需要将编解码参考代码中的模型文件 model.bin 放置到同一路径下。

-bb xxx，--bed_bitrate xxx：用于指定编码器所用声床编码比特率，取值为 xxx，码率单位为 bps。未指定此参数时，声床编码比特率默认为该声床配置下的最大编码比特率。可选码率请参考 Audio Vivid 标准中各模式的码率表。

-ob xxx，--object_bitrate xxx：用于指定编码器所用对象编码比特率，取值为 xxx，码率单位为 bps。未指定此参数时，对象编码比特率默认为 144kbps。可选码率请参考 Audio Vivid 标准中对象编码的码率表。

-d xxx.txt，--debug xxx.txt：配置时，转换工具日志会输出到 xxx.txt 文件中，若未配置，则打印日志到屏幕。

ADM 转换工具的示例命令行如下：

bw64_to_av3a input.wav out.wav metadata.bin -a out.av3a -e ./avs3Encoder -d log.txt

其中，输入 ADM 母带文件为 input.wav，转换后音频文件为 out.wav，转换后元数据二进制文件为 metadata.bin，编码后 Audio Vivid 码流文件为 out.av3a，编码器路径为./avs3Encoder（编码器依赖的模型文件 model.bin 需放在编码器路径下），输出日志文件为 log.txt。

7. 双耳渲染

7.1 功能说明

Audio Vivid 双耳渲染参考代码是对 Audio Vivid 解码后的音频数据进行渲染处理，渲染后的双声道数据用于耳机播放。双耳渲染算法为流式处理，循环从解码器输出接口接收一帧数据进行渲染，输出一帧渲染处理后的双声道数据。

双耳渲染下载链接 1：AUDIO Vivid 双耳渲染器参考代码_M

双耳渲染下载链接 2：AUDIO Vivid 双耳渲染器参考代码_Z

本部分介绍 Audio Vivid 双耳渲染器动态库/静态库的编译和算法库的使用示例。

7.2 编译方式

1.Linux 端编译

Step1: 进入 Linux 构建目录

```
cd platform_build/linux
```

Step2: 基础编译

```
./build.sh
```

Step3: 指定配置编译

```
./build.sh --build-type Debug --architecture x86_64
```

Step4: 批量编译

```
./build_all.sh
```

Step5: 音频处理测试

```
./test_audio_processing.sh
```

输出文件： - libmlslabs_binaural_render.so - 动态库 - libmlslabs_binaural_render.a - 静态库 - test_render - 测试程序。

2.Windows 端编译（MSVC）

1) 脚本编译

Step1: 进入 Windows 构建目录

```
cd platform_build\windows
```

Step2: 默认编译（Release x64）

```
.\build.ps1
```

Step3: 指定配置编译

```
.\build.ps1 -BuildType Debug -Architecture Win32
```

Step4: 批量编译所有配置

```
.\build_all.ps1
```

Step5: 测试所有编译配置

```
.\test_build_configurations.ps1
```

2) 手动编译

Step1: 创建构建目录

```
mkdir build
cd build
```

Step2: 生成 Visual Studio 项目

```
cmake .. -G "Visual Studio 17 2022" -A x64
```

Step3: 编译项目

```
cmake --build . --config Release
```

输出文件： - mlslabs_binaural_render.dll - 动态库 - mlslabs_binaural_render_static.lib - 静态库 - test_render.exe - 测试程序

3) Mac OS 端编译

Step1: 进入 macOS 构建目录

```
cd platform_build/mac
```

Step2: Apple Silicon 编译

```
./build.sh
```

Step3: Intel 编译

```
./build.sh --architecture x86_64
```

Step4: 通用二进制文件

```
./build_all.sh --architectures arm64,x86_64
```

输出文件： - libmlslabs_binaural_render.dylib - 动态库 - libmlslabs_binaural_render.a - 静态库

4) Android 端编译

Step1: 进入 Android 构建目录

```
cd platform_build/android
```

Step2: 单架构编译

```
./build.sh --abi arm64-v8a
```

Step3: 多架构批量编译

```
./build_all.sh --abis arm64-v8a,armeabi-v7a
```

注意： 如果在 Linux 环境下编译 Android 动态库，需要确保源码文件使用 Linux 换行符：

find . -type f \(-name "*.cpp" -o -name "*.h*" -o -name "*.c" -o -name "*.cc" -o -name.cxx" -o -name "*.sh" -o -name "*.mk" -o -name "*.cmake" -o -name "*.txt" \) ! -path "./build/*" -print0 | xargs -0 -P $(nproc) dos2unix

5) IOS 编译

Step1: 进入 iOS 构建目录

cd platform_build/ios

Step2: 基础编译（设备版）

./build.sh

Step3: 模拟器版本

./build.sh --platform SIMULATOR64

Step4: 通用 Framework（设备+模拟器）

./build_all.sh --create-universal

输出文件： - BinauraRender.framework - iOS Framework - BinauraRender-Universal.framework - 通用 Framework

7.3 双耳渲染接口使用说明

1.接口调用流程

调用双耳渲染 SDK 的 API 实现渲染过程的流程图如图 4 所示，渲染 SDK 的 API 接口函数定义在 include/render.h 头文件中。

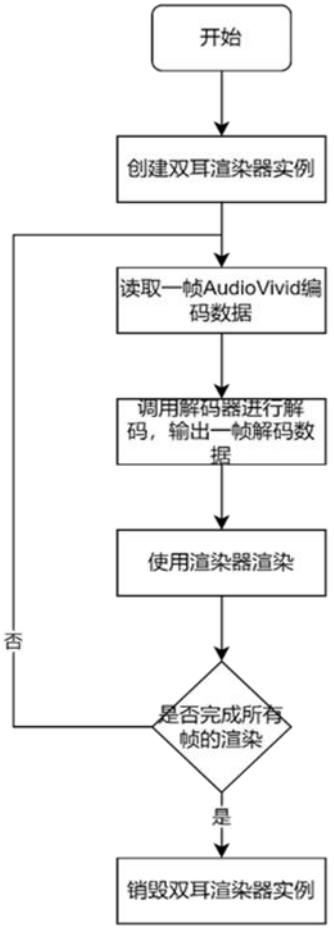

图 4 双耳渲染 SDK 渲染流程图

2.API 函数说明

1) 创建渲染器实例

Error RenderCreate(RenderHandle **render,

const RenderConfig *const config,

const Avs3MetaDataStatic *const metadata);

输入参数：

- render：渲染器实例句柄指针
- config：渲染器配置结构体指针
- metadata：静态元数据结构体指针（可选，传 NULL）

返回值：

- RENDER_SUCCESS (0)：成功
- 其他非 0 值：失败

配置结构体说明：

```
typedef struct RenderConfig {
    SampleFormat sample_format;                 // 样本格式
    int sample_rate;                            // 采样率
    int frame_len;                              // 帧长度
    bool output_binaural;                       // 双耳输出标志
    VividInputLayout input_channel_layout; // 输入通道布局
    OutputLayout output_channel_layout;      // 输出通道布局
        } RenderConfig;
```

2) 发送音频帧到渲染器

```
Error RenderSendFrame(RenderHandle *render,
                              const Avs3MetaDataDynamic *const metadata,
                              const uint8_t *const buf);
```

输入参数：

- render：渲染器实例句柄
- metadata：动态元数据结构体指针（可选，传 NULL）
- buf：输入音频缓冲区指针

返回值：

- RENDER_SUCCESS (0)：成功
- 其他非 0 值：失败

3) 接收渲染后的音频数据

```
Error RenderRecieveFrame(RenderHandle *render,
uint8_t *buf);
```

输入参数：

- render：渲染器实例句柄
- buf：输出音频缓冲区指针

返回值：

- RENDER_SUCCESS (0)：成功
- 其他非 0 值：失败

4) 销毁渲染器实例

Error RenderDestroy(RenderHandle **render);

输入参数：

- render：渲染器实例句柄指针

返回值：

- RENDER_SUCCESS (0)：成功
- 其他非 0 值：失败

5) 获取输出通道数

int RenderGetOutputChannelsNum(OutputLayout outputLayout);

输入参数：

- outputLayout：输出通道布局

6) 获取版本信息

const char *RenderGetVersion(void);

输入参数：无。

返回值：

- 返回版本字符串指针

7) 获取双耳渲染库的版本信息

const char* BinauralRenderGetVersion(void);

输入参数：无。

返回值：

返回一个指向版本字符串的指针。

8) 加载渲染器动态库

int LoadRenderDLL(const char *dllPath);

输入参数：

- dllPath：动态库路径

返回值：

- 0：成功
- （-1）：失败

9) 卸载渲染器动态库

void UnloadRenderDLL(void);

8. 扬声器渲染

8.1 功能说明

扬声器渲染 SDK 用于对 Audio Vivid 解码后的音频数据进行渲染，渲染后的多声道数据用来送到多扬声器重放。扬声器渲染 SDK 为流式处理，循环从解码器 SDK 输出接口接收一帧数据，进行渲染，输出一帧渲染处理后的多声道数据。本文档对应扬声器渲染 SDK 版本为 1.0.0。

扬声器渲染下载链接 1：AUDIO Vivid 扬声器渲染器参考代码_M

扬声器渲染下载链接 2：AUDIO Vivid 扬声器渲染器参考代码_MB

扬声器渲染下载链接 3：AUDIO Vivid 扬声器渲染器参考代码_S

8.2 编译方式

1.Script 基本用法

./compile-self-speaker.sh -h

用法: ./compile-self-speaker.sh [-t BUILD_TYPE] [-p PLATFORM] [-a ABI]

-t BUILD_TYPE	构建类型(Release/Debug/RelWithDebInfo/MinSizeRel)，默认: Release
-p PLATFORM	平台(mac/linux/windows/android/ios/ios-sim)，自动检测为当前系统
-a ABI	指令集架构:
	android: arm64-v8a, x86_64 (默认: arm64-v8a)
	mac: arm64, x86_64 (默认: arm64)
	linux: x86_64, x86 (默认: x86_64)
	windows: x86_64, x86 (默认: x86_64)
	ios: arm64 (默认: arm64)
	ios-sim: arm64, x86_64 (默认: x86_64)
-h	显示此帮助信息

2.Mac 电脑环境（可编译 Mac，IOS，Android）

1) 环境配置

a) mac 端 brew install cmake

b) 安装 Android NDK

下载地址：

https://github.com/android/ndk/wiki/Unsupported-Downloads

版本：r21e 以上

设置环境变量：

export ANDROID_NDK=${your_ndk_path}

2) 编译

使用终端执行：

cd [扬声器渲染仓库目录]

比如：

./compile-self-speaker.sh -p ios -a arm64

./compile-self-speaker.sh -p mac -a arm64

./compile-self-speaker.sh -p mac -a x86_64

./compile-self-speaker.sh -p android -a arm64-v8a

./compile-self-speaker.sh -p android -a x86_64

3) 输出

build 目录下有不同架构下产物，图 5 是 Android 端 arm64-v8a 产物目录，products 文件夹下是构建产物。其中 include 下面的 mlslabs_self_speaker_render 文件夹是对外的头文件目录。libs 文件夹下是 so 动态库，默认去除了符号。

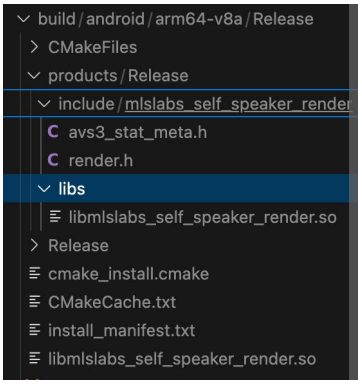

图 5 Android 端 arm64-v8a 产物目录

3.扬声器渲染接口调用说明

1) 扬声器渲染创建接口函数

本接口函数用于创建渲染器实例，接口函数定义如下：

RENDER_EXTERN

Error RenderCreate(RenderHandle **render,

const RenderConfig *const config,

const Avs3MetaDataStatic *const medadata);

输入参数：

-render：指向渲染器实例句柄的指针的指针，用于将创建的渲染器句柄传递出去；

-config：渲染器配置参数结构体；

-medadata：静态元数据结构体指针，定义于 Audio Vivid 解码器头文件 avs3_stat_meta.h 中；

返回参数：

成功返回 RENDER_SUCCESS(0)

错误返回对应错误码，返回错误码定义于 render.h，见附录 A.2 接口函数返回错误码；

2) 扬声器渲染执行发送数据接口函数

本接口函数用于发送音频数据和动态元数据给渲染器，循环调用该接口函数实现对一帧数据的发送，接口函数定义如下：

RENDER_EXTERN

Error RenderSendFrame(RenderHandle *render,

const Avs3MetaDataDynamic *const metadata,

const uint8_t *const buf);

输入参数：

-render：渲染器实例句柄，渲染器句柄来自接口函数 RenderCreate 的输入参数 render。

-metadata：动态元数据结构体指针，定义于 Audio Vivid 解码器头文件 avs3_stat_meta.h 中；

-buf：输入音频数据 buffer 指针，多通道情况下为交织排列；

返回参数：

成功返回 SPEAKER_RENDER_SUCCESS(0)

错误返回对应错误码，返回错误码定义于 render.h，见附录 A.2。

3) 扬声器渲染执行接收数据接口函数

本接口函数用于销毁创建的渲染器实例，释放内存单元，接口函数定义如下：

RENDER_EXTERN

Error RenderRecieveFrame(RenderHandle* render, uint8_t* buf);

输入参数：

-render: 指向渲染器实例句柄的指针，渲染器句柄来自接口函数 RenderCreate 的输入参数 render。

-buf: 输出音频数据 buffer 指针；

返回参数：

成功返回 SPEAKER_RENDER_SUCCESS(0)；

错误返回对应错误码，返回错误码定义于 render.h，见附录 A.2。

4) 扬声器渲染销毁接口函数

本接口函数用于销毁扬声器渲染实例，接口函数定义如下：

RENDER_EXTERN

Error RenderDestroy(RenderHandle** render);

输入参数：

-render：指向渲染器实例句柄的指针，渲染器句柄来自接口函数 RenderCreate 的输入参数 render。

返回参数：

成功返回 SPEAKER_RENDER_SUCCESS(0)

错误返回对应错误码，返回错误码定义于 render.h，见附录 A.2。

5) 扬声器渲染输出通道数接口函数

本接口函数用于返回扬声器渲染输出通道数，接口函数定义如下：

RENDER_EXTERN

int RenderGetOutputChannelsNum(OutputLayout outputLayout);

输入参数：

-outputLayout：渲染器输出扬声器布局枚举类型，定义于扬声器渲染库头文件 render.h，详见附录 A.1；

返回参数：

输出通道数

6) 扬声器渲染 SDK 版本获取接口函数

本接口函数用于返回扬声器渲染 SDK 版本，接口函数定义如下：

RENDER_EXTERN

const char* RenderGetVersion(void);

输入参数：无

返回参数：

输出扬声器渲染 SDK 版本，返回字符数组。例如：“1.0.0”代表版本 1.0.0。

9. 附录

9.1 缩略语

下列术语和定义适用于本文件：

ADM 音频定义模型（Audio Definition Model）

AI 人工智能（Artificial Intelligence）

API 应用程序编程接口（Application Programming Interface）

AVS 数字音视频编解码技术标准（Audio Video coding Standard）

DASH 基于 HTTP 的动态自适应流（Dynamic Adaptive Streaming over HyperText Transfer Protocol）

DAW 数字音频工作站（Digital Audio Workstation）

ES 基本流（Elementary Stream）

FOA 一阶立体声场信号（First Order Ambisonics）

HOA 高阶立体声场信号（Higher Order Ambisonics）

MOV QuickTime 文件格式（QuickTime File Format）

MP4 MPEG-4 第 14 部分（Moving Picture Experts Group -4 Part 14）

MSVC Microsoft Visual C++

NDK 原生开发工具包（Native Development Kit）

PCM 脉冲调制编码（Pulse-Code Modulation）

SDK 软件开发工具包（Software Development Kit）

TS MPEG-2 传输流（Moving Picture Experts Group -2 Transport Stream）

9.2 附录 A 声床布局枚举类型和错误码 （规范性）

A.1.输出声床布局枚举类型

输出声床布局用于描述音频渲染器输出设备的声道配置。其枚举定义如表 A.1 所示。

表 A.1 输出声床布局枚举类型列表

枚举定义名称	声道配置
"output_layout_mono"	Single channel
"output_layout_stereo"	Two channels
"output_layout_3_0_0"	3.0 multi-channel configuration
"output_layout_5_1_0"	5.1 multi-channel configuration
"output_layout_5_1_2"	5.1 with 2 height channels
"output_layout_5_1_4"	5.1 with 4 height channels
"output_layout_7_1_0"	7.1 multi-channel configuration
"output_layout_7_1_2"	7.1 with 2 height channels
"output_layout_7_1_4"	7.1 with 4 height channels
"output_layout_unknow"	Unknown or unsupported configuration

A.2.接口函数返回错误码

接口函数在执行完毕后会返回预定义的错误码，具体含义如表 A.2 所示。

表 A.2 接口函数返回错误码值表

序号	名称	返回码值:
RENDER_SUCCESS	成功	0
RENDER_MALLOC_ERROR	内存分配错误	-1
RENDER_HAS_NO_DST_LAYOUT	输出 layout 参数错误	-2
RENDER_HAS_NO_SRC_LAYOUT	输入 layout 参数错误	-3
RENDER_PACKFORMATID_ERROR	packFormatID 解析错误	-4
RENDER_CHANNELS_MISS_ERROR	元数据输入通道数与实际通道数不符	-5
RENDER_SPEAKER_LABEL_ERROR	speaker label 解析错误	-6

续表 A.2 接口函数返回错误码值表

序号	名称	返回码值：
RENDER_POSITION_ERROR	音源坐标超过范围	-7
RENDER_EMPTY_POINTER_ERROR	空指针错误	-8

参考文献

[1] T/UWA 009.1-2023《三维声技术规范 第 1 部分：编码分发与呈现》世界超高清视频产业联盟标准，2023.

[2] T/UWA 009.2-2-2025《三维声技术规范 第 2-2 部分：应用指南 媒体格式》世界超高清视频产业联盟标准，2025.

UHD World Association
世界超高清视频产业联盟

联系我们：
UWA联盟邮箱：support@theuwa.com
UWA联盟官网：www.theuwa.com